A Deep Learning Project to complete the missing pixels in an image using DCGAN.

Introduction:

A DCGAN is an extension of GAN, except that it explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively.

1. The Discriminator is made up of strided convolution layers, batch normalization layers, and LeakyReLU activations. The input is an image and the output is a scalar probability that the input is from the real data distribution.
2. The Generator is made up of convolutional-transpose layers, batch normalization layers, and ReLU activations. The input is a latent vector, z, that is drawn from a standard normal distribution and the output is the generated image. The strided conv-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image.

   The goal of training the discriminator D(x) is:
   1. Maximize D(x) for every image from the true data distribution x~p_data.
   2. Minimize D(x) for every image not from the true data distribution x$\nsim$ p_data.

   where p_data is the unknown distribution over the images.

   The goal of training the generator G(z) is to produce samples that fool D. The output of the generator is an image and can be used as the input to the discriminator. Therefore, the generator wants to maximize D(G(z)), or equivalently minimize 1−D(G(z)) because D is a probability estimate and only ranges between 0 and 1.

$$\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}} \log D(x) + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$$

Approach:

There are 2 types of information that can be got from an image:

1. **Contextual information:** The missing pixels can be inferred based on information provided by surrounding pixels. To keep the same context as the input image, make sure the known pixel locations in the input image *y* are similar to the pixels in *G*(*z*). We need to penalize *G*(*z*) for not creating a similar image for the pixels that we know about. Formally, we do this by element-wise subtracting the pixels in *y* from *G*(*z*) and looking at how much they differ.

2. **Perceptual information:** The information is provided by other pictures generated. To recover an image that looks real, we need to make sure that the discriminator is properly convinced that the image looks real.

```
self.contextual_loss = tf.reduce_sum(
    tf.contrib.layers.flatten(
        tf.square(tf.multiply(self.mask, self.G) - tf.multiply(self.mask, self.image))), 1)
self.adversarial_loss = self.d(self.G, training=False)
self.complete_loss = (0.999)*self.contextual_loss + (0.001)*self.adversarial_loss
self.grad_complete_loss = tf.gradients(self.complete_loss, self.zhat)
```

From the code snippet shown, you can see the contextual loss which is used for penalizing G(z) and adversarial loss (the perceptual loss) variable used to penalize the discriminator D(x).

We are finally ready to find *z*^ with a combination of the contextual and perceptual losses which is the Complete loss value shown in the code snippet above.

Step 1: Interpreting images as samples from a probability distribution

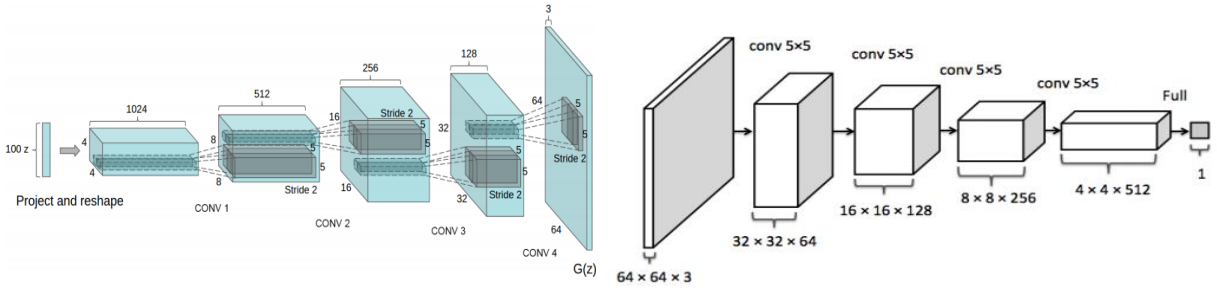
The key relationship between images and statistics is that **we can interpret images as samples from a high-dimensional probability distribution.** The probability distribution goes over the pixels of images. With images, unlike with the normal distributions, we don't know the true probability distribution and we can only collect samples.

Step 2: Generating fake images

Learning to generate new samples from an unknown probability distribution. Instead of learning how to compute the probability density function, we can learn how to generate new (random) samples with a generative model. Now that we have a simple distribution we can easily sample from, we'd like to define a function G(z) that produces samples from our original probability distribution and use this G(z) to produce fake images.

The discriminator network D(x) takes some image x on input and returns the probability that the image x was sampled from the (unknown) distribution over our images. This is where our images are sampled from. In DCGANs, D(x) is a traditional convolutional network.

Structure of generator G(z) with a DCGAN and the discriminator convolutional network D(x) respectively



Step 3: Finding the best fake image for image completion

To do completion for some image y something reasonable that doesn't work is to maximize D(y) over the missing pixels. This will result in something that's neither from the data distribution (p_data) nor the generative distribution (p_g). What we want is a reasonable projection of y
onto the generative distribution.
Now all we need to do is find some G(z^) that does a good job at completing the image. To find z^, let's revisit our goals of recovering contextual and perceptual information from the beginning of this report and pose them in the context of DCGANs to find their respective losses.

Implementation:

Algorithm:

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

Procedure:

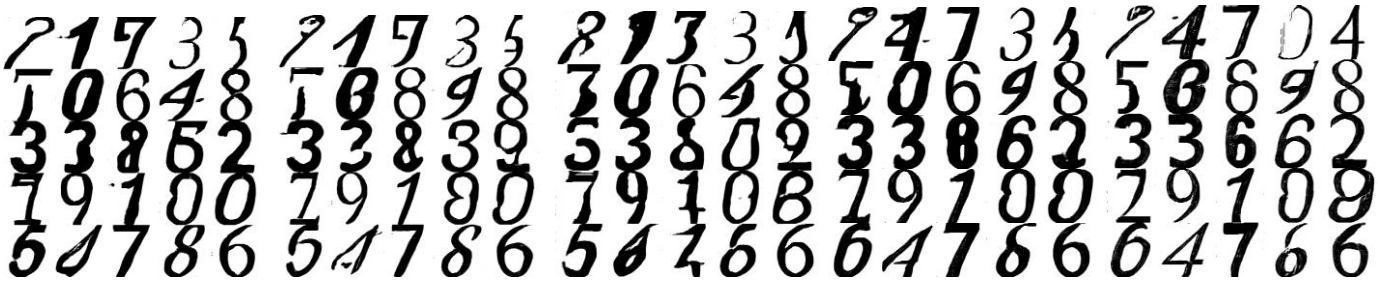
1. **Converting dataset into a tfrecords :**

   create_tfrecords.py is used to generate tfrecords from the directory of input images.

2. **Training the data :**

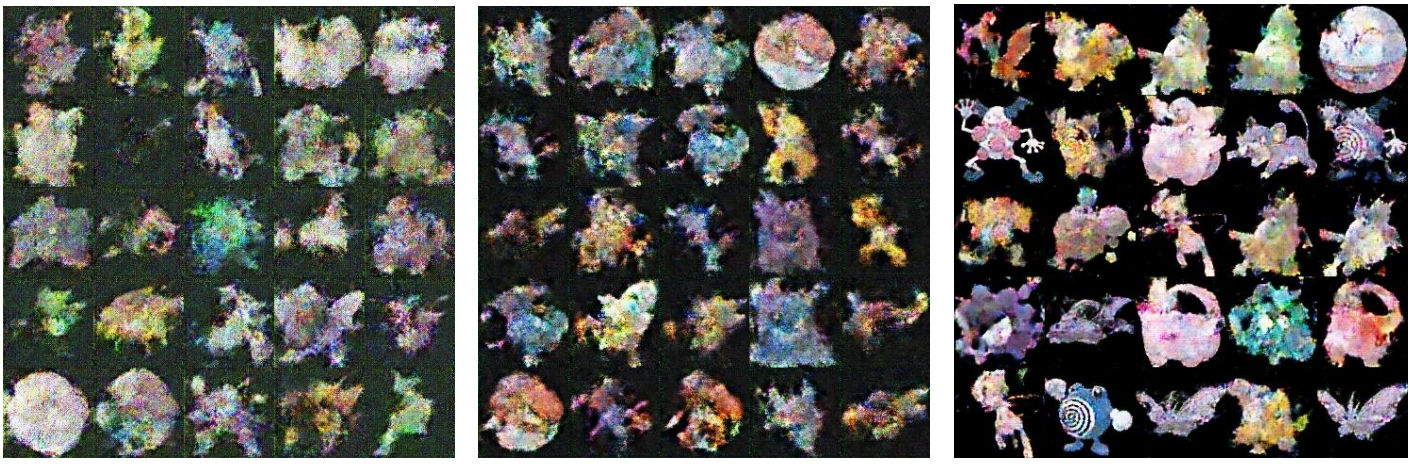
   In the file dcgan.py we have included three classes, namely, Generator, Discriminator and DCGAN. The architecture of the Generator and the Discriminator is the same as the two images in Step 2. main.py loads the training data from the tfrecords. An instance of the class DCGAN from dcgan.py is created here. The DCGAN is now trained with the help of the training data. The training was stopped at 50k iterations (for Chars74K dataset) and at 1 lakh iterations (for Pokémon dataset). The DCGAN can now be used to generate new images based on the training data. Checkpoints are made every 2000 iterations and subsequent images show how the images generated get better over time.

   Training (in iterations) on Chars74K Dataset:



10k 20k 30k 40k 50k

Training (in iterations) on Pokémon Dataset:



10k 40k 50k

82k

100k

3. Testing :

imageModule.py contains functions related to image processing like reading, cropping and saving an image. We now take an input image from the user in order to test. A mask is placed at the center of the image. The Generator is used to sample from the probability distribution in order to generate an image which is close to the input image with the mask. Suppose G(z) is the input image and G(z1) is the generated image, we run gradient descent and modify z1 for 1000 iterations in order to get G(z1) to be close to G(z). Now this generated image G(z1) is used to fill the mask in the input image.

Testing on Chars74K Dataset:

Original Image Corrupted Image (Masked original image) After 500 After 1k iterations Completed Image

Testing on Pokémon Dataset:

Original Image | Corrupted Image (Masked Original Image) | After 500 | After 1k Completed Image

Note: Google Colab was used to train and test the dataset using their GPU support for TensorFlow version 1.13.2.

Conclusion:

The missing pixels in the image were filled by using DCGANS, by selecting the best fake image generated by the Generator.

References:

- Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434.
- Yeh, Raymond A., et al. "Semantic image inpainting with deep generative models." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.
- https://github.com/sugyan/tf-dcgan
- https://www.tensorflow.org/tutorials/generative/dcgan